NumPy Array Math

Main Points

1. Some background information
2. Quick review of creating arrays
3. Perform math with arrays and scalar objects
4. Perform math with multiple arrays at the same time (element-by-element)
5. Use *NumPy* functions on all objects in an array

1 Background

Much of the power inherent in *NumPy* is the ability to perform math operations with arrays, not create them.

- Scalar values (single values that are not in an array) can be used to operate on arrays of any size
  - Scale all values up (multiply) or down (divide) by a specific factor
  - Add or subtract an offset to all values in an array
- Two arrays of the same size can operate on or with each other in an element-by-element manner
- *NumPy* is also used to perform linear algebra and matrix operations, including...
  - Inverting
  - Transposing
  - Finding determinants
  - Multiplying arrays
  - Use for finding solutions to sets of linear equations
- This document will concentrate on the following...
  - Scalar-array operations
  - Element-by-element (array-array) operations
  - Introducing functions for the creation of random numbers (both as scalars and arrays)
  - Some built-in functions for analyzing arrays
- Another document will focus on linear algebra and matrix operations

2 *NumPy* Array Review

A previous document showed how to create *NumPy* arrays using `np.array()`, `np.arange()`, and `np.linspace()`. It also demonstrated how to index, slice, and modify arrays.

Importing *NumPy* and creating arrays review

```
>>> import numpy as np
>>> np.set_printoptions(precision=5, suppress=True)

>>> np.array([3, 5, 9, 13, 42])
array([ 3,  5,  9, 13, 42])

>>> np.arange(2, 21, 2)
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])

>>> np.linspace(0, np.pi, 6)
array([0.     , 0.62832, 1.25664, 1.88496, 2.51327, 3.14159])
```

3 Scalar-Array Operations

- Math operations can be performed with arrays and scalars
- These include...
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Exponentiation
- Example: the same value can be added to or multiplied by every object in an array
- This functionality is referred to as broadcasting

The code block below performs the following math operations with a 1D array named x and a scalar named y; $x + y$, $x \times y$, $x \div y$, $y \div x$, and x^y.

1D Array-scalar operations – addition and multiplication

```
>>> x = np.arange(1, 12, 2)
>>> x
array([ 1,  3,  5,  7,  9, 11])
>>> y = 1.5

>>> x + y
array([ 2.5,  4.5,  6.5,  8.5, 10.5, 12.5])

>>> x * y
array([ 1.5,  4.5,  7.5, 10.5, 13.5, 16.5])
```

1D Array-scalar operations – division and exponentiation

```
>>> x / y
array([0.66667, 2.     , 3.33333, 4.66667, 6.     , 7.33333])

>>> y / x
array([1.5    , 0.5    , 0.3    , 0.21429, 0.16667, 0.13636])

>>> x**y
array([ 1.     ,  5.19615, 11.18034, 18.52026, 27.     , 36.48287])
```

Below a two-dimensional array named `A` is used to perform the following array-scalar operations; $y \times A$, $A \times 10$, $A + 10$, $100 - A$, and A^2.

Array-scalar operations – 2D array

```
>>> A = np.array([[2, 5, 7, 0], [10, 1, 3, 4], [6, 2, 11, 5]])

>>> y * A
array([[ 3. ,  7.5, 10.5,  0. ],
       [15. ,  1.5,  4.5,  6. ],
       [ 9. ,  3. , 16.5,  7.5]])

>>> A * 10
array([[ 20,  50,  70,   0],
       [100,  10,  30,  40],
       [ 60,  20, 110,  50]])

>>> A + 10
array([[12, 15, 17, 10],
       [20, 11, 13, 14],
       [16, 12, 21, 15]])

>>> 100 - A
array([[ 98,  95,  93, 100],
       [ 90,  99,  97,  96],
       [ 94,  98,  89,  95]])

>>> A**2
array([[  4,  25,  49,   0],
       [100,   1,   9,  16],
       [ 36,   4, 121,  25]])
```

4 Array-Array Math Operations

- Arrays of the same size can be used to perform mathematical operations on/with each other
- Can be called "element-by-element" or "item-by-item" operations
- Operations can be performed on arrays of the same size
  - Use the + and – operators for addition and subtraction
  - Use the * and / operators for multiplication and division
  - Raise the items in one array to the power of the items in another array using the ** operator

Array-array (element-by-element) math operations

```
>>> vectA = np.array([8, 5, 4], float)   # force array dtype as float
>>> vectB = np.array([10, 2, 7], float)

>>> vectA + vectB
array([18.,  7., 11.])

>>> vectA * vectB
array([80., 10., 28.])

>>> vectB / vectA
array([1.25, 0.4 , 1.75])

>>> vectA**vectB
array([1.07374e+09, 2.50000e+01, 1.63840e+04])
```

More array-array math

```
>>> A = np.linspace(0, 15, 5)
>>> A
array([ 0.  ,  3.75,  7.5 , 11.25, 15.  ])

>>> B = np.arange(10, 1, -2)
>>> B
array([10,  8,  6,  4,  2])

>>> A + B
array([10.  , 11.75, 13.5 , 15.25, 17.  ])

>>> A - B
array([-10.  ,  -4.25,   1.5 ,   7.25,  13.  ])

>>> B / A
array([    inf, 2.13333, 0.8    , 0.35556, 0.13333])
```

The results so far have likely matched expectations. However, that may or may not be the case when trying to perform mathematical operations on arrays that are the same size but different shapes. The following code block creates array C that has a single column instead of a single row. This array is no longer one-dimensional from *NumPy's* perspective because it requires two values to describe the size; it is a 5×1 array. Adding C and B yields results that are unlike the previous code blocks. All the values in B are added to each of the values in C; creating a row for each item in C. This is a good example of broadcasting.

Array-array operations on arrays with different shapes

```
>>> C = np.array([1, 3, 4, 6, 7]).reshape(5,1)
>>> C
array([[1],
       [3],
       [4],
       [6],
       [7]])

>>> B + C
array([[11,  9,  7,  5,  3],
       [13, 11,  9,  7,  5],
       [14, 12, 10,  8,  6],
       [16, 14, 12, 10,  8],
       [17, 15, 13, 11,  9]])
```

The following code block an array named x with a range of integers from 1 to 8 inclusive is created. The calculation $x^2 - 4x$ is completed using the array x and the result is assigned to the variable y and then printed.

An array-array math example

```
>>> x = np.arange(1, 9)
>>> y = x**2 - 4*x
>>> print(y)
[-3 -4 -3  0  5 12 21 32]
```

Next, two arrays a and b are created such that a has 9 equally spaced values between 1 and 9 and b has 9 equally spaced values between 0.25 and 0.5. The calculation $\frac{5a\,b^{1.2}}{a-2b}$ is completed using arrays a and b and the resulting array is assigned to the name c before it is printed.

An array-array math example

```
>>> a = np.linspace(1, 9, 9)
>>> b = np.linspace(0.25, 0.5, 9)
>>> c = 5*a*b**(1.2)/(a - 2*b)
>>> print(c)
[1.89465 1.51811 1.56404 1.67635 1.81296 1.96207 2.11902 2.28156 2.44842]
```

****** START HERE *******

5 Using Math Functions with Arrays

- When using mathematical functions on numeric arrays **do not** use the `math` module
- Functions (and constants) in the `math` module are designed to work on scalar values, not arrays of values
- *NumPy* includes its own mathematical functions and constants that are designed to work with arrays
    - Use `np.pi` not `math.pi`
    - Use `np.sin()` not `math.sin()`
- The following table shows some common *NumPy* math functions that match up to `math` module functions
- It is assumed that the statement `import numpy as np` was used to import the *NumPy* module

math	**numpy**
`math.sin(x)`	`np.sin(x)`
`math.cos(x)`	`np.cos(x)`
`math.tan(x)`	`np.tan(x)`
`math.asin(x)`	`np.arcsin(x)`
`math.acos(x)`	`np.arccos(x)`
`math.atan(x)`	`np.arctan(x)`
`math.atan2(y, x)`	`np.arctan2(y, x)`
`math.hypot(x, y)`	`np.hypot(x, y)`
`math.radians(x)`	`np.radians(x)`
`math.degrees(x)`	`np.degrees(x)`
`math.pi`	`np.pi`
`math.e`	`np.e`
`math.exp(x)`	`np.exp(x)`
`math.log(x)`	`np.log(x)`
`math.log10()`	`np.log10(x)`
`math.round(x)`	`np.round(x)`
`math.sqrt(x)`	`np.sqrt(x)`

A trig example

The following code block creates an array of angles named theta that range from 0 to $360°$ in steps of $15°$. The angle array is then used to calculate arrays x and y using $x = \cos(\theta)$ and $y = \sin(\theta)$. The tabulate module is then used to present the results.

Creating a cosine and sine table for angles

```
>>> # tabulate needs to be imported to use
>>> from tabulate import tabulate

>>> theta = np.arange(0., 361, 15)
>>> print(theta)
[  0.  15.  30.  45.  60.  75.  90. 105. 120. 135. 150. 165. 180. 195.
 210. 225. 240. 255. 270. 285. 300. 315. 330. 345. 360.]

>>> # calculate 'x' and 'y' using NumPy's cosine and sine functions
>>> # 'theta' is in degrees and needs converting to radians
>>> x = np.cos(np.radians(theta))
>>> y = np.sin(np.radians(theta))
>>> values = zip(theta, x, y)
>>> headers = ['Degrees', 'cosine ', 'sine  ']
>>> print(tabulate(values, headers, floatfmt=('3.0f','+0.4f','+0.4f')))
  Degrees    cosine     sine
---------  ---------  --------
        0    +1.0000   +0.0000
       15    +0.9659   +0.2588
       30    +0.8660   +0.5000
       45    +0.7071   +0.7071
       60    +0.5000   +0.8660
       75    +0.2588   +0.9659
       90    +0.0000   +1.0000
      105    -0.2588   +0.9659
      120    -0.5000   +0.8660
      135    -0.7071   +0.7071
      150    -0.8660   +0.5000
      165    -0.9659   +0.2588
      180    -1.0000   +0.0000
      195    -0.9659   -0.2588
      210    -0.8660   -0.5000
      225    -0.7071   -0.7071
      240    -0.5000   -0.8660
      255    -0.2588   -0.9659
      270    -0.0000   -1.0000
      285    +0.2588   -0.9659
      300    +0.5000   -0.8660
      315    +0.7071   -0.7071
      330    +0.8660   -0.5000
      345    +0.9659   -0.2588
      360    +1.0000   -0.0000
```

A few more *NumPy* math functions

```
>>> q = np.arange(1,6,dtype="float")
>>> q
array([1., 2., 3., 4., 5.])

>>> # natural log (often written as ln) of q
>>> np.log(q)
array([0.     , 0.69315, 1.09861, 1.38629, 1.60944])
```

```
>>> # square root of q
>>> np.sqrt(q)
array([1.     , 1.41421, 1.73205, 2.     , 2.23607])

>>> # e^q, can use np.e or np.exp()
>>> np.e**q
array([  2.71828,   7.38906,  20.08554,  54.59815, 148.41316])
>>> np.exp(q)
array([  2.71828,   7.38906,  20.08554,  54.59815, 148.41316])
```

6 *NumPy* Statistical Functions

- *NumPy* offers a number of functions for performing statistical analysis on values in arrays
- The following table describes the most common of these functions
- The descriptions are based on one-dimensional arrays (which are the most common)

`numpy` **Function**	**Description**
`np.sum()`	Sum the values
`np.mean()`	Arithmetic mean
`np.median()`	Median value
`np.std()`	Standard deviation
`np.var()`	Variance
`np.max()`	Maximum value
`np.argmax()`	Index of the maximum value
`np.min()`	Minimum value
`np.argmin()`	Index of the minimum value
`np.sort()`	Create a sorted copy

Using *NumPy's* statistical functions

```
>>> A = np.array([2, 5, 9, 13, 12, 10])

>>> # compute the mean
>>> np.mean(A)
8.5

>>> # compute the median
>>> np.median(A)
9.5
```

```
>>> # standard deviation
>>> np.std(A)
3.8622100754188224

>>> # maximum value and its position in the array
>>> np.max(A)
13

>>> # minimum value
>>> np.min(A)
2

>>> # add all of the values
>>> np.sum(A)
51

>>> # make a sorted copy
>>> np.sort(A)
array([ 2,  5,  9, 10, 12, 13])
```

The code block below demonstrates a few more *NumPy* statistics-related functions.

A few more statistics-related functions

```
>>> # cumulative sum
>>> np.cumsum(A)
array([ 2,  7, 16, 29, 41, 51])

>>> # product
>>> np.prod(A)
140400

>>> # cumulative product
>>> np.cumprod(A)
array([     2,     10,     90,   1170,  14040, 140400])
```

7 Random Number Generation

- Within `numpy` there is a random number module called `random`
- Use `np.random.rand()` in order to use the `rand()` function
- The following table describes a number of the random functions that are available available
- When a range is given as $[0, 1)$ it means between 0 and 1, including 0 but not including 1, and is referred to as a half-open range

Function	**Description**
`np.random.rand()`	Random float from a uniform distribution over $[0, 1)$
`np.random.rand(x)`	Array of `x` random floats from a uniform distribution over $[0, 1)$
`np.random.rand(r, c)`	An $r \times c$ array of random floats from a uniform distribution over $[0, 1)$
`np.random.randint(x)`	A random integer from $[0, x)$
`np.random.randint(low, high)`	A random integer from $[low, high)$
`np.random.randint(low, high, size)`	Array of length `size` filled with random integers from $[low, high)$
`np.random.randint(low, high, (r, c))`	An $r \times c$ array of random integers from $[low, high)$
`np.random.randn()`	A random value from a normal distribution of mean 0 and variance of 1
`np.random.randn(x)`	Array of `x` random values from a normal distribution
`np.random.randn(r, c)`	$r \times c$ array of random values from a normal distribution
`np.random.shuffle(arr)`	Randomly shuffle array `arr` in place

- Use `(low - high)*np.random.rand() + low` to generate a random floating point value between `low` and `high`.

Random number function examples

```
>>> # single random float
>>> np.random.rand()
0.41536391492679337

>>> # 5 random integers between 1 and 6, inclusive
>>> np.random.randint(1, 7, 5)
array([6, 5, 4, 4, 2])

>>> # shuffle and print all integers from 1 to 10, inclusive
>>> x = np.arange(1, 11)
>>> np.random.shuffle(x)
>>> print(x)
[ 3  6  4  5  2  1  8  9 10  7]

>>> # random integer from 1 to 100, inclusive
>>> np.random.randint(1, 101)
71
```

```
>>> # 2x5 array of random integers from 0 to 9, inclusive
>>> np.random.randint(0, 10, (2, 4))
array([[1, 9, 4, 2],
       [3, 2, 3, 6]])

>>> # 10 from random normal dist with variance = 4 and mean = 50
>>> v = 4
>>> m = 50
>>> np.random.randn(10)*4 + 50
array([47.01905, 51.73883, 58.65255, 45.07991, 48.27787, 54.57671,
       46.36795, 47.77925, 54.16571, 58.92623])
```